John von Neumann Institute for Computing

Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block Lanczos algorithm – a case study

R.H. Bisseling, I. Flesch

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 819-826, 2006.

http://www.fz-juelich.de/nic-series/volume33

Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block Lanczos algorithm — a case study

Rob H. Bisseling[a], Ildikó Flesch[b]

[a]Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands (Rob.Bisseling@math.uu.nl)

[b]Department of Information and Knowledge Systems, Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands (ildiko@cs.ru.nl)

Abstract. A case study is presented demonstrating the application of the Mondriaan package for sparse matrix partitioning to the field of cryptology. An important step in an integer factorisation attack on the RSA public-key cryptosystem is the solution of a large sparse linear system with 0/1 coefficients, which can be done by the block Lanczos algorithm proposed by Montgomery. We parallelise this algorithm using Mondriaan partitioning and discuss the high-level components needed. A speedup of 10 is obtained on 16 processors of a Silicon Graphics Origin 3800 for the factorisation of an integer with 82 decimal digits, and a speedup of 7 for 98 decimal digits.

1. Introduction

The security of the widely used RSA public-key cryptosystem [12] is based on the fact that finding the prime factors of a large integer is extremely time-consuming. The state-of-the-art in integer factorisation methods tells us how large the keys used in RSA must be to withstand attacks based on trying to find the prime factors for a given public-key value.

On May 9, 2005, Bahr, Böhm, Franke, and Kleinjung [1] announced a new record factorisation: with help of te Riele and Montgomery they factorised the 200 decimal-digit number originally posed as the RSA-200 challenge in 1991. This factorisation used the Number Field Sieve (NFS) [9], which is currently the best factorisation method for large integers. In practice, the NFS almost always finds a non-trivial factor of a composite number within a few attempts. The two most time-consuming parts of this method are the *sieving step* and the *matrix step*. The sieving step took from December 2003 to December 2004, and was done by farming out jobs to a variety of computers, taking a total of 55 CPU years (at the equivalent speed of a 2.2 GHz Opteron processor). The matrix step is more tightly coupled and needs more memory since it involves a large sparse matrix, with 64 million rows and columns and 11×10^9 nonzeros for RSA-200. Because of this, it must be carried out on a parallel computer. The matrix step took about three months on a cluster of 80 Opterons. The linear system of the matrix step was solved by a block Wiedemann algorithm [5]. An alternative method would be the block Lanczos algorithm proposed by Montgomery [10].

In the present work, we will discuss the high-level components needed in a parallel computation of the matrix step, such as Mondriaan matrix partitioning. In this paper, we focus on the block Lanczos algorithm, but the same high-level components are also needed for the block Wiedemann algorithm.

2. Sequential algorithm

2.1. Construction of the sparse matrix

The sieving step in the factorisation of a large number n tries to find many pairs (a_j, b_j) of integers such that $a_j \equiv b_j \pmod{n}$ and a_j and b_j are the product of squares and small primes. Let p_i be the ith prime, i.e., $p_1 = 2, p_2 = 3, p_3 = 5$, etc. and let $p_0 = -1$. Then we can write each a_j uniquely as a finite product

$$a_j = \prod_i p_i^{m_{ij}}. \tag{1}$$

Note that a_j is square if and only if all exponents m_{ij} are even. Define a matrix A by $a_{ij} = m_{ij} \bmod 2$. The matrix A is sparse because integers have only a limited number of prime factors. Define a similar matrix B for the integers b_j.

The matrix step tries to construct a subset of pairs $(a_j, b_j),\ j \in S$, such that $\prod_{j\in S} a_j$ and $\prod_{j\in S} b_j$ are both square. Let $\alpha^2 = \prod_{j\in S} a_j$ and $\beta^2 = \prod_{j\in S} b_j$. If $\gcd(\alpha\beta, n) = 1$, then $\gcd(\alpha - \beta, n)$ is a factor of n, and hopefully a non-trivial one. If we write $S = \{j :\ x_j = 1\}$, where $\mathbf{x}$ is an integer vector with 0/1 components x_j, we see that the two products are square if and only if $A\mathbf{x} = 0$ and $B\mathbf{x} = 0$, where all computations are carried out modulo 2, i.e., in the finite field GF(2). Let the $n_1 \times n_2$ matrix C represent the two simultaneous linear systems,

$$C = \begin{bmatrix} A \\ B \end{bmatrix}.$$

Thus, we need to solve $C\mathbf{x} = 0$. Figure 1 shows an example matrix C.

2.2. Sequential block Lanczos algorithm

To find (part of) the nullspace $\mathcal{N}(C)$ of C, we can apply the block Lanczos algorithm as proposed by Montgomery [10]. Since this algorithm is only suitable for symmetric matrices, it is applied to C^TC in such a way that it finds a nullspace $\mathcal{N}(C^TC)$ that is as large as possible. There is no need to form the product C^TC explicitly: multiplication by C^TC is carried out as multiplication by C followed by multiplication by C^T. Furthermore, it suffices to store only C. Since $\mathcal{N}(C) \subset \mathcal{N}(C^TC)$, we hope to be able to find some vector $\mathbf{x} \in \mathcal{N}(C)$; this can be done by a postprocessing procedure [10] after the block Lanczos algorithm.

The block Lanczos algorithm is applied to solve $C^TCX = C^TCY$, where X and Y are $n_2 \times N$ matrices. The solution matrix X contains a set of N columns, each representing a solution $\mathbf{x}$. This way, we obtain N solutions in one run of the algorithm. The size of N is chosen as the word size of an integer on the computer used, say $N = 32$. This choice enables storage of the complete matrix X in a single one-dimensional integer array of length n_2. It also allows the use of efficient bit operations in the algorithm. The matrix Y is chosen as a random bit matrix. The aim of this approach is to obtain many independent solutions of $C^TC\mathbf{x} = 0$, given by the columns of the matrix $X - Y$.

Algorithm 1 summarises the steps of the block Lanczos algorithm. At the first occurrence of a matrix in the algorithm, its size is given as a superscript. Matrix subscripts denote the order in a sequence of matrices. All multiplication operations are explicitly shown by using an asterisk, e.g. $C * V$ on line 15. The result matrix is then written as CV. There is no need to store both V and V^T; only one of the two matrices suffices. The matrix $Cond$ represents the termination condition; SS^T represents the index set S. The function generateWS generates new matrices W^{inv} and SS^T. It only involves computations on small $N \times N$ matrices. Its description is omitted for the sake of brevity; see [10, Fig. 1] for details.

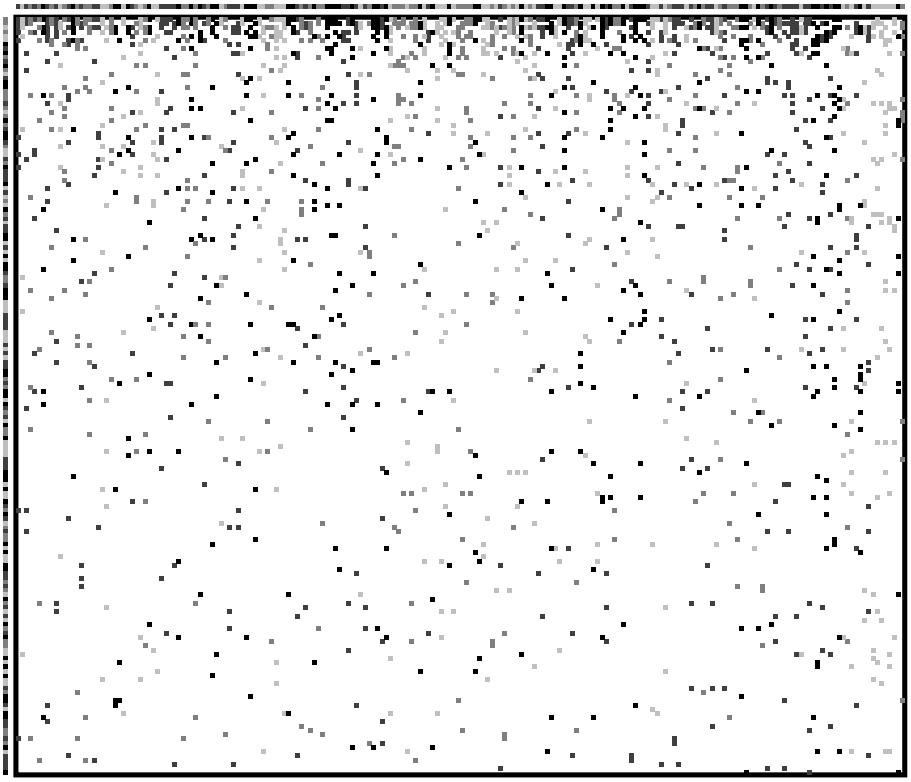

Figure 1. Matrix C corresponding to a sparse linear system $C\mathbf{x} = 0 \pmod 2$ obtained by the Multi-Polynomial Quadratic Sieve (MPQS) method for a 30-decimal integer. The 179×210 matrix C has 1916 nonzero elements. Each matrix row represents a prime; each column a pair of integers (a_j, b_j). The primes are sorted in increasing order, where each prime may occur at most twice. The matrix has been partitioned for four processors of a parallel computer, shown in different grey shades, by using the Mondriaan package [14]. Also shown is a partitioning of the input vector (above the matrix) and output vector (left) for a sparse matrix–vector multiplication $\mathbf{y} = C\mathbf{x}$. Matrix source: courtesy of Richard Brent.

3. High-level components for parallel computation

The matrix step of the block Lanczos algorithm requires the following major high-level components:

- sparse matrix–vector multiplication;
- sparse matrix partitioning;
- vector partitioning;
- dense vector inner-product computation;
- AXPY operation;
- global-local indexing mechanism.

Algorithm 1 The sequential block Lanczos algorithm.

Input: matrices C of size $n_1 \times n_2$ and Y of size $n_2 \times N$.
Output: matrix X, such that $C^T(CX) = C^T(CY)$.
1. Initialise:
1a. $W_{-2}^{\text{inv}}{}^{N \times N} = W_{-1}^{\text{inv}}{}^{N \times N} = 0$
1b. ${V_{-2}}^{n_2 \times N} = {V_{-1}}^{n_2 \times N} = 0$
1c. ${CV_{-1}}^{n_1 \times N} = 0$
1d. ${K_{-1}}^{N \times N} = 0$
1e. ${SS^T}_{-1}^{N \times N} = I_N$
1f. $X^{n_2 \times N} = 0$
2. ${V_0}^{n_2 \times N} = C^T * (C * Y)$
3. ${CV_0}^{n_1 \times N} = C * V_0$
4. ${Cond_0}^{N \times N} = (CV_0)^T * CV_0$
5. $i = 0$
while $Cond_i \neq 0$ **do**
7. $[W_i^{\text{inv}}, SS_i^T] = \text{generateWS}(Cond_i, SS_{i-1}^T, N, i)$
8. $X = X + V_i * (W_i^{\text{inv}} * (V_i^T * V_0))$
9. $C^T CV_i^{n_2 \times N} = C^T * CV_i$
10. $K_i = ((CV_i)^T * (C * (C^T CV_i))) * SS_i^T + Cond_i$
11. $D_{i+1}^{N \times N} = I_N - W_i^{\text{inv}} * K_i$
12. $E_{i+1}^{N \times N} = -W_{i-1}^{\text{inv}} * (Cond_i * SS_i^T)$
13. $F_{i+1}^{N \times N} = -W_{i-2}^{\text{inv}} * (I_N - Cond_{i-1} * W_{i-1}^{\text{inv}}) * K_{i-1} * SS_i^T$
14. $V_{i+1} = C^T CV_i * SS_i^T + V_i * D_{i+1} + V_{i-1} * E_{i+1} + V_{i-2} * F_{i+1}$
15. $CV_{i+1} = C * V_{i+1}$
16. $Cond_{i+1} = (CV_{i+1})^T * CV_{i+1}$
17. $i = i + 1$
18. Return X.

These components can be viewed as building blocks that occur in many different applications; for instance, they occur in both the block Lanczos algorithm and the block Wiedemann algorithm for the matrix step, but also in most iterative linear system solvers.

Sparse matrix–vector multiplication involving the matrix C occurs on lines 2, 3, 9, 10, 15 of Algorithm 1. Here, the sparse bit matrix C is multiplied by an $n_2 \times N$ bit matrix, which can be viewed as N multiplications by a bit vector of length n_2, or the transpose matrix C^T is multiplied. The parallel component required is a four-phase algorithm, consisting of: (i) communication of the components of the input vectors to exactly those processors that need them; (ii) local matrix–vector multiplication; (iii) communication of local results to the owner of the corresponding output vector component; (iv) and finally addition of these results. For more details, see [2, Chap. 4]. This parallel algorithm for sparse matrix–vector multiplication improves upon that used by Montgomery [11] in his parallel version of the block Lanczos algorithm because in our approach the communication exploits the sparsity of the matrix C; in [11], however, the amount of communication is as large as for a dense matrix. The algorithm should work for every possible distribution of the matrix and the input and output vectors. Note that the algorithm is a generalisation of the regular sparse matrix–vector multiplication to the multi-vector case.

Sparse matrix partitioning can be done by any of the available sparse matrix partitioners based on multilevel hypergraph partitioners, such as hMetis [7], Mondriaan [14], Par$k$way [13], PaToH [4], or Zoltan [6]. Par$k$way and Zoltan are able to do the partitioning itself in parallel. In the present work, we used the sequential partitioner Mondriaan.

Vector partitioning can be done by algorithms that try to balance the communication load of the sparse matrix–vector multiplication. Such a partitioning is incorporated in the Mondriaan package, and an improved version is described in [3]. Note that in the block Lanczos algorithm we have two types of vectors: those of length n_1 and those of length n_2. The two types can be partitioned independently, taking the result of the preceding matrix partitioning into account.

Dense vector inner-product computation occurs on lines 4, 8, 10, 16. It is easiest to perform if all vectors of the same length have the same distribution. The vector partitioning in Mondriaan does not take the number of components assigned to each processor into account, although in practice this number is not too badly balanced among the processors. A possible extension would be to perform the vector partitioning for multiple objectives, including balancing the inner-product computation.

The *AXPY operation* ('A times X Plus Y') is a well-known level 1 operation [8] from the Basic Linear Algebra Subprograms (BLAS). In iterative linear system solvers, it has the form $\mathbf{y} := \alpha\mathbf{x}+\mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors and α is a scalar. Its double-precision version is sometimes called DAXPY. In Algorithm 1, the AXPY operation occurs on lines 8 and 14. For instance, on line 8, we can view the multiplication of the $n_2 \times N$ bit matrix V_i by an $N \times N$ matrix as the multiplication of an integer vector of length n_2 by a small object, analogous to a scalar. An AXPY is carried out in parallel by replicating the scalar so that every processor has a copy and letting each processor multiply the local part of the vector by the scalar.

A *global-local indexing mechanism* is needed at the start of the block Lanczos algorithm. After the matrix and vectors have been distributed, the processors know which matrix elements and vector components they own, but they do not know where to obtain the input vector components they need in the matrix–vector multiplication, or where to send contributions for output vectors. The solution to this problem is that the global address of every vector component is first stored at a location that can be inspected by all processors. This location is called the *notice board* in BSPedupack [2], or the *data directory* in Zoltan [6], which provides many additional services besides partitioning. For example, the address of component x_j with global index j of a vector $\mathbf{x}$ can be found at processor $j \bmod p$ at local index $\lfloor j/p \rfloor$, where p is the number of processors. After the address has been retrieved at the start of the block Lanczos algorithm, the current value of vector component x_j can be obtained from that address each time it is needed.

All other parts of Algorithm 1 are less important. Matrices of size $N \times N$ are small (only N integers) and can easily be communicated and replicated. For instance, the computations of lines 7, 11, 12, 13 are carried out redundantly by every processor.

4. Numerical experiments

We performed numerical experiments on up to 16 processors of the Silicon Graphics Origin 3800 at SARA in Amsterdam. We used two matrices, `c82` and `c98a`, produced by the MPQS method during the factorisation of composite integers with 82 and 98 decimal digits, respectively. For problems of this size, MPQS is faster than NFS. The matrices originating in MPQS are similar to those of NFS and representative of the wider class of sieving matrices. The properties of these matrices are given in Table 1.

We implemented the parallel block Lanczos algorithm using the high-level components described in Section 3. We partitioned the two matrices and the corresponding vectors using the Mondriaan

Name	n_1	n_2	Nonzeros
c82	16307	16338	507716
c98a	56243	56274	2075889

Table 1
The properties of the two test matrices.

package [14] version 1. The execution times of the parallel program with $p = 1$ for c82 and c98a are about 80 s and 1200 s, respectively. We only have a parallel version of the program available, so we cannot use a sequential version to compare with. Thus, there will be some overhead in our reference version, which is the parallel program run on one processor. The overhead mainly consists of global-local index transformations and unnecessary calls to the synchronisation mechanism. This overhead is expected to be small, because the index transformations are carried out in a preprocessing step, and thus are removed from the main loop of the computation. Furthermore, the main loop contains only a few synchronisations. For $p = 1$, all communications reduce to memory copies. The relative speedup of the parallel program compared to the $p = 1$ case is given in Figures 2 and 3. Note that we achieve a higher speedup on the smaller problem, which is unusual, and which we find hard to explain. Certainly, cache effects must play a role here. (We have partially optimised our implementation to make it cache friendly.) The speedups achieved are reasonable, but not optimal. One reason for this is that the vector partitioning should be improved. The current tests used the vector partitioning of Mondriaan version 1; for version 2, we expect better results.

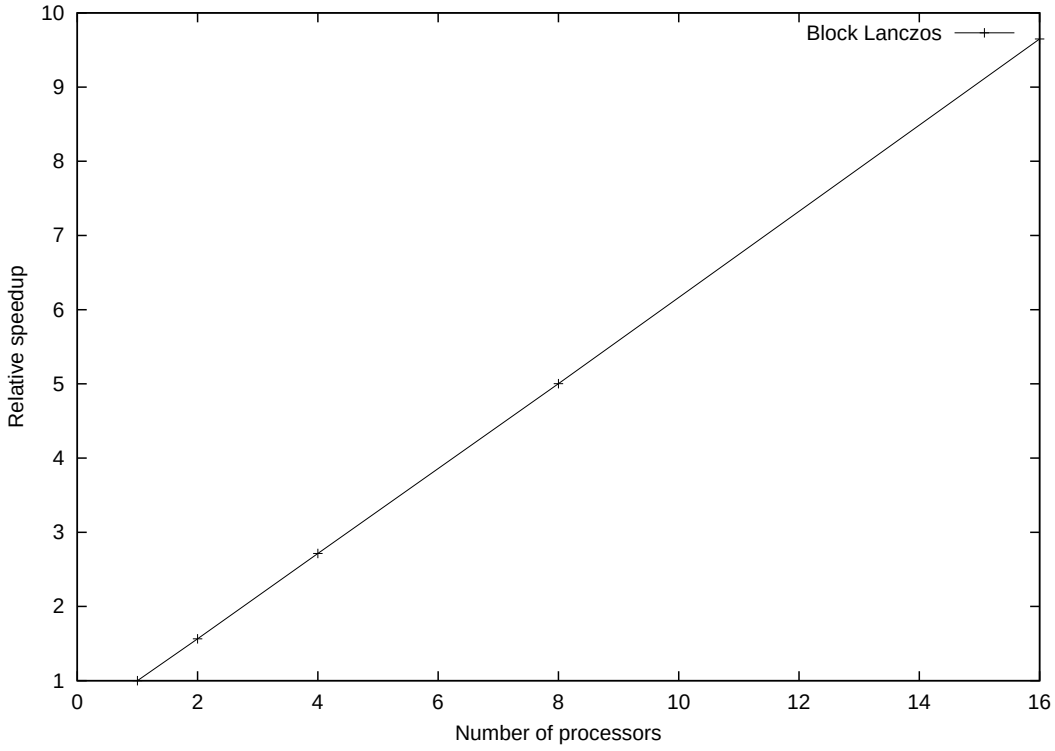

Figure 2. Speedup of parallel block Lanczos algorithm for test matrix c82.

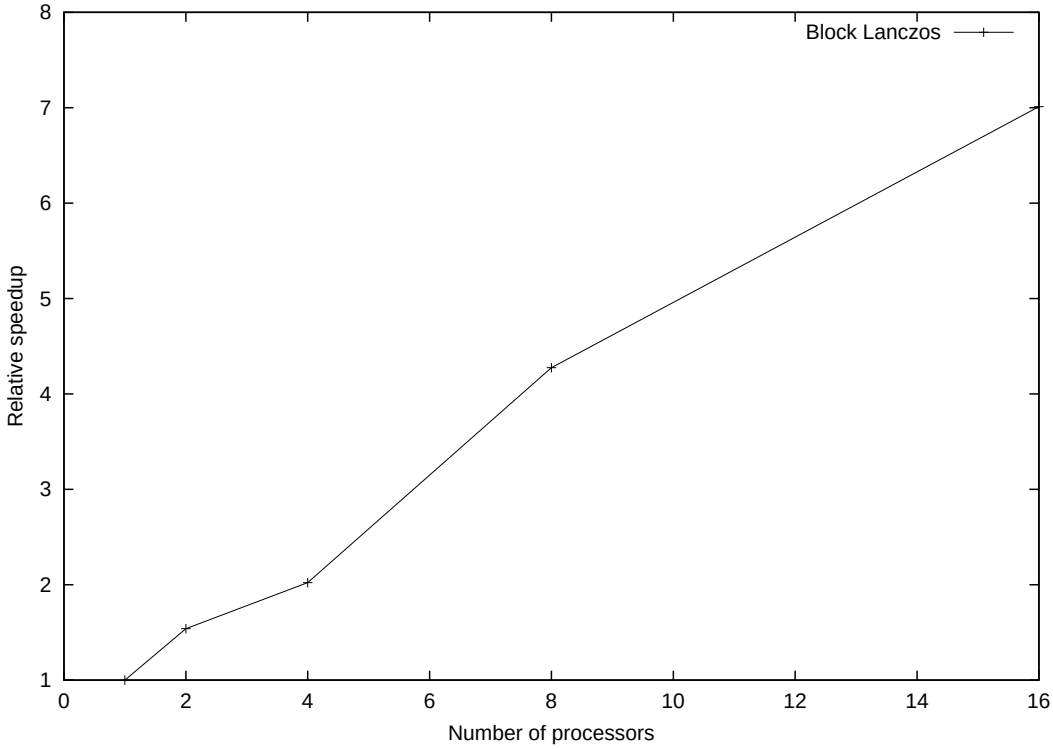

Figure 3. Speedup of parallel block Lanczos algorithm for test matrix `c98a`.

5. Conclusions and future work

We have studied an application in the field of cryptology, the solution of sparse linear systems in the binary field GF(2). We have identified important high-level components for this application and discussed their parallel aspects. This application has some particular characteristics (e.g. the computations modulo 2), but otherwise it stands for a much larger class of applications such as iterative methods for the solution of linear systems and eigensystems. The identified high-level components are important for this whole class.

An issue that also emerged from this application is that we cannot balance the computational load completely by preprocessing to find good matrix and vector partitionings. If we make use of the current bit pattern in the vectors and in the small $N \times N$ matrices (with $N = 32$) to avoid certain unnecessary operations, we save work but we also introduce a dependence of the work load on the current state. This may lead to load imbalance. It is a challenge to find a dynamic procedure to mitigate this effect.

Much research is carried out these days on higher-level tools for parallelisation. The high-level components identified here could provide focus for these efforts. If the tools would help in developing efficient and flexible components for the block Lanczos algorithm, this would have an impact on a wide range of applications.

Acknowledgements

We thank Richard Brent for providing the test matrices used in this paper and for many valuable suggestions. We thank Fatima Abu Salem for interesting and helpful discussions on integer factorisation. We also thank Herman te Riele for useful comments on the initial version of this paper. Part of the research has been funded by the Dutch BSIK/BRICKS MSV1-2 project. Computer time has been partially funded by the Dutch National Computer Facilities foundation (NCF).

References

[1] Friedrich Bahr, M. Böhm, Jens Franke, and Thorsten Kleinjung. Factorisation of RSA-200. Announcement, http://www.loria.fr/ zimmerma/records/rsa200, May 9, 2005.

[2] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, Oxford, UK, March 2004.

[3] Rob H. Bisseling and Wouter Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005. Special Issue on Combinatorial Scientific Computing.

[4] Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[5] Don Coppersmith. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, 1994.

[6] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, March/April 2002.

[7] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings 36th ACM/IEEE Conference on Design Automation*, pages 343–348. ACM Press, New York, 1999.

[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[9] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In *Proceedings 22nd Annual ACM Symposium on the Theory of Computation*, pages 564–572, 1990.

[10] Peter L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In *Proceedings EUROCRYPT'95*, volume 921 of *Lecture Notes in Computer Science*, pages 151–168. Springer-Verlag, Berlin, 1995.

[11] Peter L. Montgomery. Parallel block Lanczos. In *Proceedings RSA-2000*, 2000.

[12] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[13] Aleksandar Trifunovic and William J. Knottenbelt. A parallel algorithm for multilevel k-way hypergraph partitioning. In *Proceedings Third International Symposium on Parallel and Distributed Computing, Cork, Ireland*, July 2004.

[14] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.